Aalto University
School of Science
Master's Degree Programme in Security and Mobile Computing

José Bernardo Víquez Zamora

IoT Application Provisioning Service

Master's Thesis
Espoo, July 31, 2017

Supervisors: Mario Di Francesco, Aalto University
Danilo Gligoroski, NTNU
Instructor: Pranvera Kortoçi

Aalto University
School of Science
Master's Degree Programme in Security and Mobile Computing

ABSTRACT OF MASTER'S THESIS

Author:	José Bernardo Víquez Zamora		
Title:	IoT Application Provisioning Service		
Date:	July 31, 2017	**Pages:**	51
Major:	Security and Mobile Computing	**Code:**	T3011
Supervisors:	Mario Di Francesco, Aalto University Danilo Gligoroski, NTNU		
Instructor:	Pranvera Kortoçi		

Constant development of software requires updating our Internet of Things (IoT) devices regularly. Some services such as transportation, health care, surveillance and electronic payments require high availability, even during a software update.

IoT updates in urban scenarios require connectivity based on the Internet Protocol (IP) and long range connection with adequate speed. Normally, these requirements are provided by cellular network (i.e., using a SIM card) to connect to the Internet. This option presents several disadvantages: it is very expensive and it exposes IoT devices to security threats due to the permanent connection to the Internet. These challenges could be addressed by leveraging long-range broadcast communication (e.g., FM broadcast). IoT devices periodically listen for and receive updates through such a communication infrastructure, without actually being connected to the Internet.

This thesis presents a system to provide software updates for IoT devices through long-range broadcast communication technologies. A prototype has been developed based on the concept of "seamless updates". This allows performing software updates in the background, hence ensuring the availability of a device during the installation time of an update. This seamless update process was implemented on an embedded device (i.e., a Raspberry Pi 3) with a Linux-based operating system. Furthermore, a web-based backend has been implemented. Such a backend allows IoT developers to upload their updates targeting a specific class of devices and schedule when the update will be sent. The security goals of integrity and authentication are accomplished by signing the updates in the backend and verifying it at the IoT device. Moreover, a performance evaluation is performed for the system upgrade service with different parameters to sign the updates.

Keywords:	IoT, provisioning service, Over-the-air programming, OTA, seamless updates, software upgrades, IoT security
Language:	English

Acknowledgements

First of all, I would like to thank God for giving me the opportunity to study abroad and accomplish my dream of getting a master's degree.

Secondly, I would like to thank the NordSecMob Programme for giving me the opportunity to be part of such an inspiring and multinational programme, broaden my knowledge in the security field, and develop personally and professionally. Gratitude goes to the Ministry of Science, Technology and Telecommunications of Costa Rica for granting me a scholarship that made it possible to attend such a programme. I highly value all the help and support I got from the study coordinators and the staff of Norwegian University of Science and Technology (NTNU) and Aalto University during these two years.

More specifically, I thank Mario Di Francesco for guiding me during my master's thesis work over the past months: thank you Mario for all your lessons, patience and support. In addition, I would like to give thanks to Pranvera Kortoçi, my advisor, for all the support and advice during this period. I would also like to thank the people of the Computer Science Department for helping and encouraging me every day while writing this thesis. Special thanks go to Maria, Mariusz and Gopika.

Furthermore, I would like to thank my friends in Finland that supported and were with me in the good and the bad times. Particularly, my flatmates Klaudia and Nadin, with whom I shared this journey. In addition, I would like to give credit and show gratitude to the friends I made in Norway who always stayed connected and cheered me up, sharing life even from distance. Big thanks to Kasia, Nicole, Simone and Verena who visited me during my period in Finland and brought me a lot of joy.

Last but certainly not least, I would like to thank my family and friends from Costa Rica: Muchas gracias a toda mi familia por su apoyo incondicional durante estos dos años de maestría, sinceramente que he sido bendecido por Dios al tener una familia tan maravillosa.

Le doy gracias a mi papá por su fortaleza y apoyo a pesar de su enfermedad. A mi mamá por sus constantes frases de apoyo y sus oraciones. A

mi hermano German José por su soporte y guía en los momentos difíciles. A mi cuñada Gisela por sus consejos y muestras de afecto. A mi hermano Ricardo por sus videollamadas y buenos consejos sobre vida social y deporte. A mi cuñada Stefany por su constante colaboración ante cualquier problema y su cariño. Además, quisiera agradecer especialmente a mi hermana Ana Marcela por apoyarme, ayudarme, soportarme y por siempre presente, al igual que a mi cuñado Rafa que me ha brindado su apoyo y soporte durante todo este tiempo desde Holanda.

También quisiera darles las gracias a mis sobrinos por sus muestras de cariño y afecto. Cada video, foto o mensaje me hicieron sentir cerca, a pesar de la distancia, pero además me impulsaron a seguir adelante. Le agradezco a Fabián por sus enseñanzas sobre *Clash Royale* para mis momentos de distracción; a Diego por sus comentarios graciosos que siempre me alegran el día; a Matías por enseñarme sobre seguridad ("shoulder surfing") cuando con 4 años se aprendió mi patrón para desbloquear el teléfono y por ser una persona con la que puedo compartir mi obsesión por *Pokémon*; a Ignacio por sus sonrisas y ocurrencias que le alegran la vida a cualquiera; a Paula por su ternura, sus besos y frases de cariño que me levantaban el ánimo cada vez que me sentía mal y a Elena, mi chinita hermosa, por darme la bendición de ser tío una vez más de éste lado del mundo, fue una experiencia increíblemente gratificante. ¡Los amo demasiado!.

Además, quiero agradecer a mi abuelita hermosa por su ejemplo y sus oraciones, a mis tíos y tías por su apoyo, y especialmente a mis primos y primas incondicionales que desde diferentes partes del mundo se mantuvieron pendientes de mi vida, apoyándome, guiándome y mandando sus mensajes de cariño. Al pertenecer a una familia de mucho más de 50 parientes, se me hace difícil escribir todos los nombres; sin embargo, quiero darle gracias especiales a María Isabel y a Ivonne quienes a pesar de la distancia me han acompañado constantemente en ésta travesía.

Por último, quiero agradecer a todos mis amigos de la infancia, colegio, universidad, trabajo y de la vida quienes se han hecho presentes por medio de mensajes, llamadas, palabras de aliento y ayuda. ¡Son increíbles!.

Los llevo a cada uno en el corazón y les deseo las mayores bendiciones en la vida. Sin ustedes ésta maestría no se hubiera logrado.

In summary, thank you, kiitos, takk, muchas gracias and pura vida!

Espoo, July 31, 2017

José Bernardo Víquez Zamora

Abbreviations and Acronyms

AWS	Amazon Web services
ANATEL	National Telecommunication Agency in Brazil
DoS	Denial of Service
DSA	Digital signature algorithm
DTV	Digital television receivers
DV	Digital video
ECDSA	Elliptic curve digital signature algorithm
FICORA	Finnish Communications Regulatory Authority
FM	Frequency modulation
FS	File system
GB	Gigabytes
GUI	Graphical user interface
HSM	Hardware secure module
HTTPS	Hypertext transfer protocol over TLS
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of things
IP	Internet protocol
JAR	Java archive
JDK	Java development kit
MB	Megabytes
MHz	Megahertz
MITM	Man-in-the-middle
MVC	Model-view-controller
NIST	National Institute of standards and technology
NOOBS	New out of the box software
NTNU	Norwegian University of Science and Technology
OS	Operating system
OTA	Over the air programming
OWASP	Open Web Application Security Project

PHP	PHP: Hypertext Preprocessor
PIXEL	Pi Improved Xwindow Environment, Lightweight
PK^{-1}	Private key
PK	Public key
RAM	Random-access memory
RPi3	Raspberry Pi 3
RSA	Ron Rivest, Adi Shamir and Leonard Adleman algorithm
RTL	Register-transfer level
SD	Secure digital
SDR	Software defined radio
SHA	Secure hash algorithm
SQL	Structured query language
SW	Software
TCP	Transmission control protocol
TEE	Trusted execution environment
TLS	Transport layer security
TV	Television
US	United States of America
USB	Universal serial bus
UHF	Ultra high frequency
URL	Uniform resource locator
VHF	Very high frequency
WSN	Wireless sensor networks
XSS	Cross-site scripting

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter formulates the problem addressed by this thesis work. Afterwards, the chapter presents the contribution of the research, the implementation and the evaluation. Finally, the chapter details the structure of the thesis with a brief explanation of each chapter.

1.1 Problem Statement

Constant development of software requires updating our Internet of Things (IoT) devices regularly. Some services such as transportation, health care, surveillance and electronic payments require high availability, even during a software update.

Typically, IoT updates in urban scenarios require connectivity based on the Internet Protocol (IP) and long range connection with adequate speed. Normally, these requirements are provided by cellular network (i.e., using a SIM card) to connect to the Internet. This option presents several disadvantages: it is very expensive and it exposes IoT devices to security threats due to the permanent connection to the Internet. In addition to these disadvantages, the IoT devices possess non-updatable software requiring physical manipulation to update them and normally during the update process the IoT device becomes unavailable to the users.

These challenges could be addressed analyzing the current infrastructure in the cities to broadcast digital data, the latest hardware and software solutions in terms of wireless software updates, as well as the security recommendation of international organizations (e.g., NIST and OWASP) to provide a wireless, secure, wide and efficient upgrade service for IoT devices.

1.2 Contribution

In this thesis work, a solution for an IoT application provisioning service is presented leveraging long-range broadcast communication technology with a defined components, architecture and security requirements. Additionally, a prototype has been developed based on the concept of "seamless updates" to provide a system upgrade service for embedded devices. This system allows performing software updates in background, hence ensuring the availability of a device during the installation time of an update. Furthermore, a Web Application has been implemented to provide an interface for IoT application developers to upload their device-specific software updates. The implementation applies digital signature security for the update files to provide integrity and authentication leveraging an Oracle library normally used by Android to sign their applications. The thesis includes a performance evaluation of the prototype and its different digital signature algorithms, based on security recommendations.

1.3 Structure

The rest of the thesis is organized as follows. Chapter 2 introduces the background about the current scenarios on IoT provisioning services and long-range broadcast communications technology, as well as the computer security recommendations for these services. Chapter 3 presents the design of the proposed solution of an IoT application provisioning service, its components and use cases. Chapter 4 details the implementation of a seamless updates prototype on an embedded device, the development of a Web Application to upload software updates and a performance evaluation of the system upgrade service. Finally, Chapter 5 concludes this thesis with suggestions for future work.

Chapter 2

Software updates for embedded devices

This chapter describes the main concepts related to software provisioning systems. Firstly, the chapter provides some examples of existing software updates based on over the air programming. Moreover, it introduces the concept of seamless updates for embedded devices, focusing in IoT devices. Subsequently, the chapter reviews the transmission of data over long-range broadcast technology. Finally, the chapter presents the security provided by the current software upgrade systems.

2.1 Over the air programming

Over the air programming (OTA) refers to different mechanisms to update the software in embedded (IoT) devices wirelessly, hence requiring no physical access to reprogrammed devices [8]. These mechanisms include updating applications, modifying the configuration of devices or exchanging security keys. Currently, OTA supports software upgrades for several devices including set-top boxes, sensors and mobile phones. Thus, this upgrade process enables the manufacturers to improve the software, extend its functionalities and fix security vulnerabilities by providing new software versions instead of requiring physical maintenance of the devices.

To explain the OTA process, we review three different OTA application scenarios:

- **Wireless sensor networks.** This scenario provides communication by some multi-hop short-range technology such as ZigBee. OTA in WSN distributes the update by some gateways near end devices. Hence,

transmission of data is restricted by the distance between nodes. For example, in WSN using ZigBee in an urban scenario the sensors or actuators should be positioned less than 100 meters from each other [9].

- **Digital television (DTV) receivers.** This scenario sends software updates by leveraging television broadcast services. DTV receivers download the update packages from a broadcaster that defines the strength of the signal, operation frequency and communication channel [10]. Usually, the broadcaster requires a transmission license provided by a local authority (e.g., Finnish Communications Regulatory Authority (FICORA) in Finland and National Telecommunication Agency (ANATEL) in Brazil). Normally, receivers download the update only when the audio and video interfaces are turned off. One of the drawbacks of this solution is that the download process could be interrupted if the user turns on the DTV receiver. Moreover, DTV devices allow only partial updates of their system. However, these devices possess two updatable partitions: the current and the backup partition. This dual partition characteristic enables the device to rollback in case of a problem with the new update [10].

- **Android devices.** Android devices apply OTA to distribute new versions of their software and to update the applications installed in their platform. However, this requires the devices to be connected to the Internet. Typically, these devices are polling OTA servers for new updates, which provide the download Uniform Resource Locator (URL) and information to display to the user about the update. Afterwards, the update is downloaded by the device. Once the download is finished, the system verifies that the update contains a valid certificate. The approval of the user is necessary to start the installation. Subsequently, the device reboots to apply the update. Finally, the device reboots normally when the system is updated completely. During this process, the device becomes unavailable, i.e., the user waits for several minutes the end of installation and reboot processes [11].

The structure of the OTA update is different for every system. For instance, Android 5.0 systems implement block OTA updates [12]. This mechanism updates the entire partition as a block and removes the individual file check of the partition. This block creates only a binary patch and ensures that the device installed the actual update. After the update is performed, the boot process verifies that the file system of the device was not altered by a virus (e.g., malware) during the upgrade process. In case the system

detects the file system was tampered during the update execution, the verification boot fails and the device returns to its previous version. This block OTA leverages two types of updates:

- **Full update.** This method generates a complete copy of the system. This increases the size of the update, but it is easier to realize. If the size of the image is large, this approach may require long time to update the device.
- **Incremental update.** This method creates a diff binary between the current block and the updated one. This reduces the size of the images. However, it requires longer time to generate the binary.

As previously mentioned, in some cases OTA requires IoT devices to be connected wirelessly to the Internet to download their latest update. Typically, the devices access wireless Internet through cellular networks. Hence, this connectivity requirement represents higher costs (i.e., cheaper solutions are presented in [10]) for the companies to provide their devices with Internet connectivity and expose them to several Internet issues that will be addressed in Subsection 2.4.

In addition to wireless data transmission, OTA assumes reliable communications. Normally, broadcast communications are unreliable and to address this problem, it is necessary to leverage forward error correction techniques, e.g. Reed-Solomon coding of packets [13]. The main idea of this technique consists of sending a redundant number of packets to the end devices, i.e., if the whole update package length is m, the sender will send k additional redundant packets. The end device will receive at most $k + m$ packets. However, in case a packet is lost, it is possible to recover the missing information if at least m distinct packets are received. Therefore, this mechanism decreases the throughput but increments the robustness of the system in presence of packet loss [13].

2.2 Seamless Updates

The *seamless updates* concept was introduced by Google in 2016, and it is currently implemented in Android Nougat 7.0. This concept is based on the updating process of the Chrome operating system (ChromeOS). The implementation in ChormeOS requires three partitions on the hard drive of the device: one *stateful partition* for the profile logs and the home directory, as well as two partitions for the root of the file system [3]. The purpose of this solution is to execute the update installation in background, hence this technique provides additional availability to the devices.

Although seamless updates increase the availability of the devices, they are implemented only for certain Android devices because it requires two systems A and B installed in the hard drive. Each system is provided with their own group of partitions, e.g., boot, root and vendor. This group of partitions are also known as slots (slot A and slot B). While the system is running in slot A, a process known as daemon runs in the background[1]. If an update is found, the process installs it in slot B. Furthermore, each slot contains a variable that stores the number of attempts the bootloader tries to boot with the slot[2]. Once the installation is finished, the process adds some value greater than zero (e.g., 1, 2 or more) to the attempts variable of slot B. Hence, next time the device boots the system tries to run slot B with the installed update. If the boot process is successful, slot B is set as bootable and the attempts variables are cleared for the rest of the slots. Otherwise, slot B is set as non-bootable after the attempts are over and the system tries to boot from a slot set as bootable. Figure 2.1 shows the update process of ChromeOS [3].

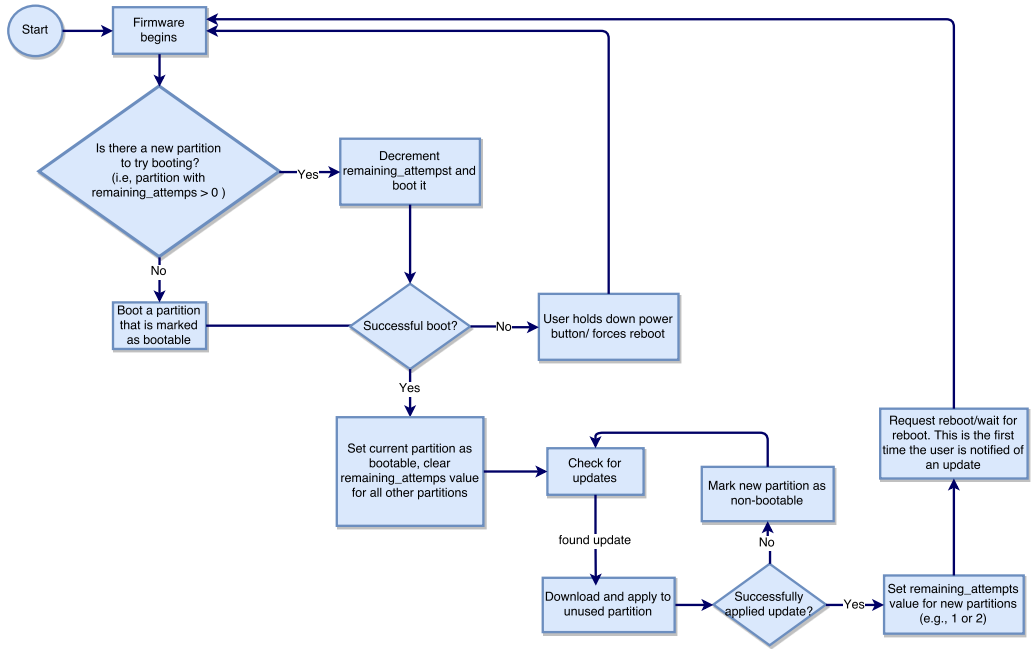


Figure 2.1: Autoupdate process of ChromeOS [3].

If neither of the systems is bootable, then the bootloader starts a recovery process to reinstalled the system (i.e., ChromeOS reinstallation is showed in

[1] In Android is called *update_ engine* daemon [14]
[2] In Android is called *remaining_ attemps*

Figure 2.2). This process requires a recovery image saved in a USB drive or SD card [4].

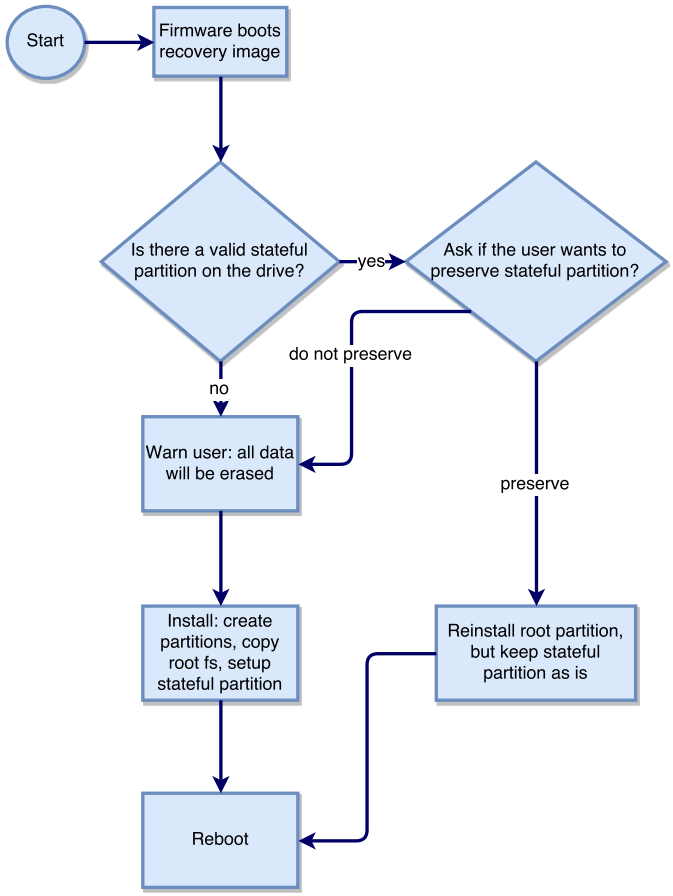


Figure 2.2: Reinstallation process of ChromeOS [4].

This approach provides robustness to the devices in case the updating process fails and also allows the device to continue running while the installation process of the update is being performed [14].

However, one of the drawbacks of this concept is that devices require twice as much as memory space to host two systems at the same time. Google has presented a solution with a compressed file system such as SquashFS. This compress file system is more efficient than a *.tar.gz* file. When the system boots, it mounts the compressed partitions as read-only file systems. To achieve read/write access of these partitions it is necessary to utilize UnionFS, that enables the system to write over a read-only file system. These files systems are combined to save space on the drive of the device, as well as to

perform seamless updates on Android devices more efficiently [15].

2.3 Long-range broadcast data transmission

Cities around the world hold the infrastructure for long-range broadcast data transmission. The transmission of data in the radio spectrum is performed over licensed, unlicensed or mixed bands. Normally, government authorities allocate the spectrum and assign frequencies to the licensed users over a region [16]. Within the spectrum there are unused frequencies at a certain time and location, known as *whitespaces* [17]. To identify whitespaces efficiently, the radio devices are provided with cognitive radios capabilities to dynamically determine which band is available to transmit information depending on the frequency, time and geolocation [18]. To share the radio spectrum between all of its users is necessary to provide some regulation models including Licensed Shared Access (LSA), Priority Access (PA) and Authorized Shared Access (ASA) [19].

In 2011, IEEE published the first standard for wireless communications over television whitespaces. This provides access to a wide area (i.e., 20-30 km). The standard leverages frequency bands between 54 MHz and 862 MHz (i.e., VHF/UHF TV broadcast) [20]. Several studies evaluate the possibility to implement wireless broadband access on TV whitespaces (e.g., [17, 21]) or to complement existing infrastructure for mobile data delivery (e.g., [22, 23]). In addition to TV whitespaces, there are underutilized frequencies present in the frequency modulation (FM) band. One example of data transmission over FM frequencies is the timetable displays in the bus stops in Finland. In fact, their data updates are broadcasted over FM [24].

Furthermore, register-transfer level (RTL) dongles with software defined radio (SDR) capabilities provide a wide band radio scanner. Depending on their specification these dongles are able to receive signals in the range from 52 MHz to 1766 MHz [25]. These RTL-SDR dongles could be connected to IoT devices by USB and extend their functionality by receiving data wirelessly over long-range communication technologies such as TV, FM or DV (i.e., digital video) broadcast. Such a data transmission provides a cheaper solution than cellular networks to obtain their software updates and requires no Internet connection.

2.4 Computer security

Security includes several properties that are addressed in this thesis work [26] :

- **Confidentiality.** Enables secrecy of the information by allowing only authorized users to access sensitive information. Moreover, provides privacy to users, so unauthorized users cannot obtain their data.
- **Integrity.** Guarantees that the information is not altered by any unauthorized user when it is being transmitted, processed or stored.
- **Availability.** Ensures that the system is usable and accessible when an authorized user requires it.
- **Authentication.** Identifies who is performing certain actions, including the sender of information, the executor of the process or the author of the data. The authentication process should be performed each time someone tries to access resources in the system.
- **Authorization.** Grants or denies access to the resources of the system.
- **Non-Repudiation.** Establishes that users cannot deny the authorship of their actions.

All of these security properties are required to analyze the security in a system, as well as to define its security goals. For example, OTA requires evaluating the security properties to provide a successful upgrade system to the end devices. Furthermore, provisioning services support authentication and integrity. These properties are obtained by leveraging asymmetric cryptography. This type of cryptography involves a pair of keys: one private and one public. Such keys are owned by the companies developing the software update. The software update file is signed with the private key. Signatures provide integrity to the update file in case an attacker tampers the file during the upgrade process. Additionally, the update includes a certificate with the public key signed by a certified authority to identify the author of the update. Once the update is downloaded, the end device verifies that the certificate is valid and that the file is not modified since its signature [27].

Over the past years, developers implemented and tested several practices to secure IoT systems [28]. Best practices in IoT security include performing a threat analysis [29], follow security recommendations, learn from old attacks and apply design patterns [30]. More specifically, IoT systems should consider the following security advices:

- Consider physical attacks in the threat analysis [31]
- Implement automatic key management [32] and hardware randomness [33].
- Utilize only properly evaluated security protocols and key lengths [34].
- Provide a secure upgrade functionality to fix future vulnerabilities [35].
- Not utilize the same key for all the devices because if one device is compromised the complete system becomes vulnerable [35].
- Following recommendation on protocols such as Transport Layer Secure(TLS) and Datagram TLS to secure communication channels [36].

In addition to these recommendations, there are privacy considerations that should be included in the development of IoT systems. It is important to provide the users with awareness about what information is required by the system. Moreover, the users should consent or not to expose their data. Additionally, the system should secure sensitive information of the users, e.g., geolocation, photos, passwords, contacts and payment info [37].

To ensure platform security, IoT devices could be provided with a Trusted Execution Environment (TEE). TEE provides confidentiality and integrity to the system, i.e., isolating the non-trusted software, data and hardware from the trusted ones (Figure 2.3). This trusted technology comprises its own boot, operating system and hardware. Some functionalities are performed in the TEE including secure booting, authentication, payment and cryptographic operations. In addition to these functionalities, the environment stores any secret key and performs any sensitive process, e.g., execute trusted applications updated with OTA [5].

Open Web Application Security Project (OWASP) presented as number one vulnerability not securing the backend server. For this reason, a server implementation should consider OWASP risk analysis on most common vulnerabilities and recommendations to secure Web applications [38]. In addition to OWASP recommendations, a TEE could be integrated to the server to secure any sensitive data. However, TEE is not feasible in cloud solutions, the main reason is that the server could not be migrated because the TEE is integrated within the hardware. To solve this migration problem, companies such as Amazon Web Services (AWS) provide another type of secure solution feasible for cloud architecture. For example, Cloud hardware secure module (CloudHSM) stores sensitive data and performs any cryptographic operations. Furthermore, cloud solutions require a client module in the backend to communicate with the CloudHSM functionality [39].

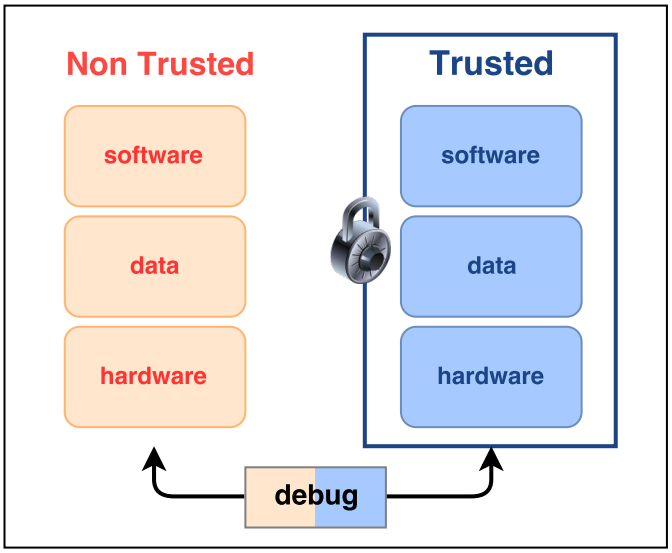


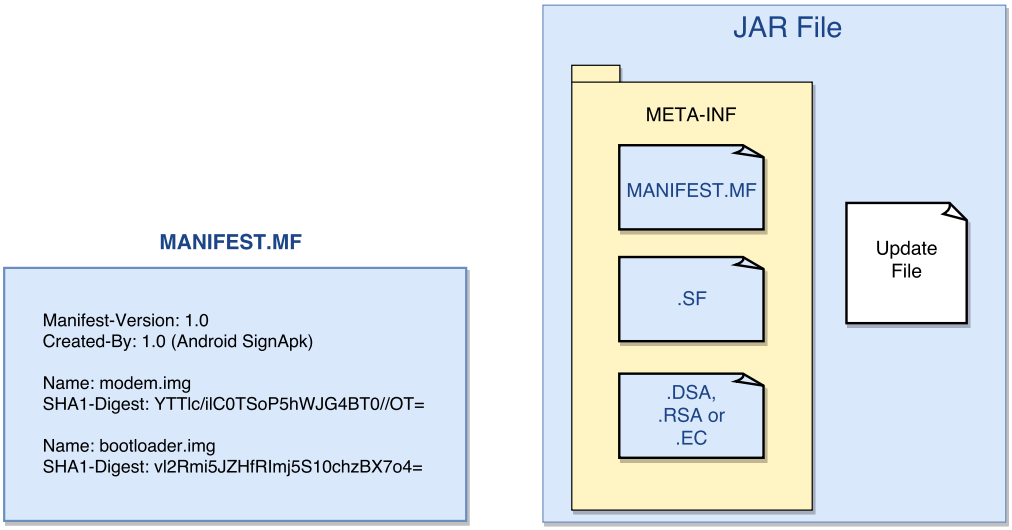
Figure 2.3: Trusted Execution Environment (TEE) overview [5].

Typically, Android OTA updates are Java ARchive (JAR) files. These JAR files contain metadata inside a *META-INF* folder. Metadata includes a *MANIFEST.MF* file with a header information about the created manifest and a list of files contained in the JAR. Each of these listed files include an output after performing a digested hash algorithm to the binary file (e.g., Figure 2.4a). The digest algorithm is created with a Secure Hash Algorithm (SHA), which takes the binary file as an input and generates an output of a specific length. The size of this output depends on the complexity of the algorithm selected by the user. If the utilized algorithm generates the same output for two different inputs it is called a collision. These collisions represent vulnerabilities of the hash algorithm [1]. Table 2.1 shows a comparison of the SHA algorithms.

Algorithm name	**Output size (bits)**	**Collision found (80 rounds)**
SHA-0	160	Yes
SHA-1	160	Yes
SHA-2	256/384/512	No

Table 2.1: Comparison of Hash Algorithms [1]

JAR files are signed to authenticate the author of the file and to assure the integrity of the file. In fact, Oracle developed a tool called *jarsigner* to sign



(a) (b)

Figure 2.4: a) Example of a manifest file in a JAR b) Example of the structure of a JAR.

these JAR files [40]. This library allows the backend to sign the JAR file with different signing algorithms: DSA (Digital Signature Algorithm), RSA (Ron Rivest, Adi Shamir, and Leonard Adleman algorithm) or ECDSA (Elliptic Curve Digital Signature Algorithm) [41]. Thus, the developer selects the signing algorithm and the desired key size. However, the JAR is not signed if the library detects security vulnerabilities in the specified signing parameters (e.g., chosen digest algorithm and key size), and the signing method returns an error. Once the JAR file is signed, a signature (i.e., *.SF* file) is added to the metadata folder and signature block file (i.e., *.DSA*, *.RSA*, or *.EC* file depending on the signature algorithm chosen) [40, 42]. Chapter 4.4 evaluates the performance of the signing algorithms DSA, RSA and ECDSA.

Chapter 3

A System for IoT Software Updates

This chapter describes a system for IoT application provisioning service using long-range broadcast communication technologies. Firstly, the reference scenario is introduced along with the main assumptions of the system. Secondly, the architecture of the system is presented with focus on its high-level components and use cases. Moreover, a threat model is defined to establish the security requirements. Finally, the hardware and software architecture of the system is explained before proceeding with the implementation explained in Chapter 4.

3.1 Reference Scenario

The reference scenario for the proposed system is a metropolitan area consisting of several heterogeneous IoT devices. There is a large number of devices in the considered urban scenario, however they belong to a relatively limited number of classes (or types). Each device type is a platform (e.g., Raspberry Pi or Edison Board) running a specific OS. Moreover, IoT devices run an OS and a set of applications which receive software updates. IoT devices have no Internet connection but are only equipped with a broadcast wireless receiver suitable for long-range communication (downlink only). These requirements represent a base for the system design. Figure 3.1 presents such a reference scenario.

The main idea of the system is to provide the IoT application developers with a platform to distribute their software updates. Figure 3.2 presents the architecture for the IoT application provisioning service. This system contains the following components:

- **Backend Server.** It is located in the Cloud, and provides an interface to the IoT app developers to create and schedule software updates.

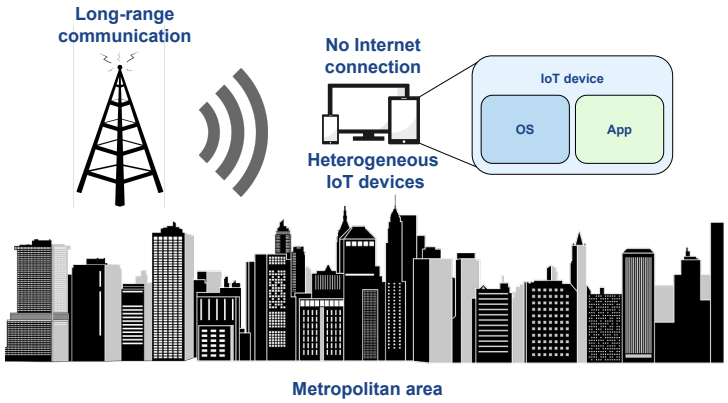


Figure 3.1: Reference scenario for the IoT Application Provisioning Service.

Additionally, the backend server is responsible for signing the update packages, as well as transmitting the packages to the radio transmitter.

- **Radio transmitter.** Acts as a gateway that communicates with the backend server by Transmission Control Protocol (TCP) and Internet Protocol (IP) to obtain the update package. Once it obtains the package, the gateway transmits the data to a long-range broadcast antenna. Consequently, the antenna broadcasts the data over the radio spectrum leveraging Radio Data System Protocol [43].

- **IoT device.** Receives the data corresponding to the update package through a long-range broadcast receiver (e.g., RTL-SDR dongle). Moreover, the broadcast receiver is directly attached to the IoT device with USB cable or some other wired/wireless technology (e.g., thunderbolt [44], wireless USB or Bluetooth [45]). Once the update package is completely received, the device verifies the signature of the package and installs the new software.

The thesis work focuses on the design of the backend server and of the supporting software running at the IoT device. Accordingly, we detail these components next.

To analyze the functionalities of the system and their interactions we present use case diagrams. Such diagrams show how the actors relate to the system functionalities. An actor represents users, external hardware, or

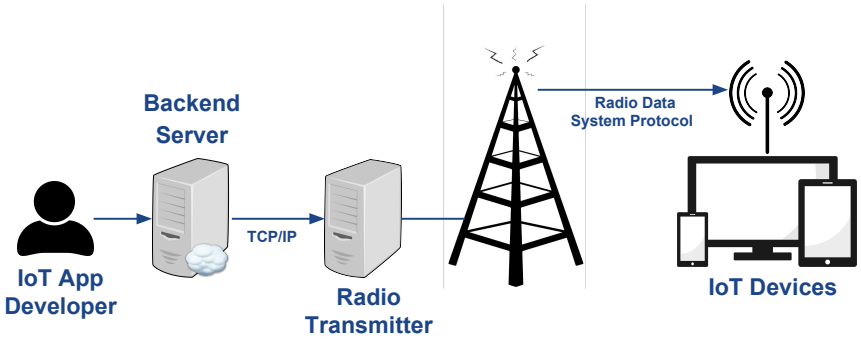


Figure 3.2: Architecture of the IoT Application Provisioning Service.

other subjects that interact with the system [46]. Only the following actors are considered:

- **IoT Application Developer.** Interacts with the system to perform certain actions including logging in, uploading the update package, modifying personal information and managing the device types. The IoT developer builds updates for a specific type of device. In fact, this actor is restricted to perform actions only on the corresponding updates, device types and profiles owned by him/her.
- **Administrator.** Possesses full access to the information stored in the backend server. After passing the authentication process to access the system, the administrator is allowed to perform several functionalities including the management of users, device types and updates.
- **Radio Transmitter.** Receives the signed update from the backend server.
- **IoT device.** Obtains the update package broadcasted from the radio transmitter. The software upgrade process starts after the IoT device receives the update image completely. Such a device requires a boot-loader and two independent OSes to implement seamless updates (see Subchapter 2.2)

Figure 3.3 contains the use cases of the backend server:

- **ManageUsers.** Includes functionalities to create, edit, view and delete users. The management of users is accessed by the IoT application

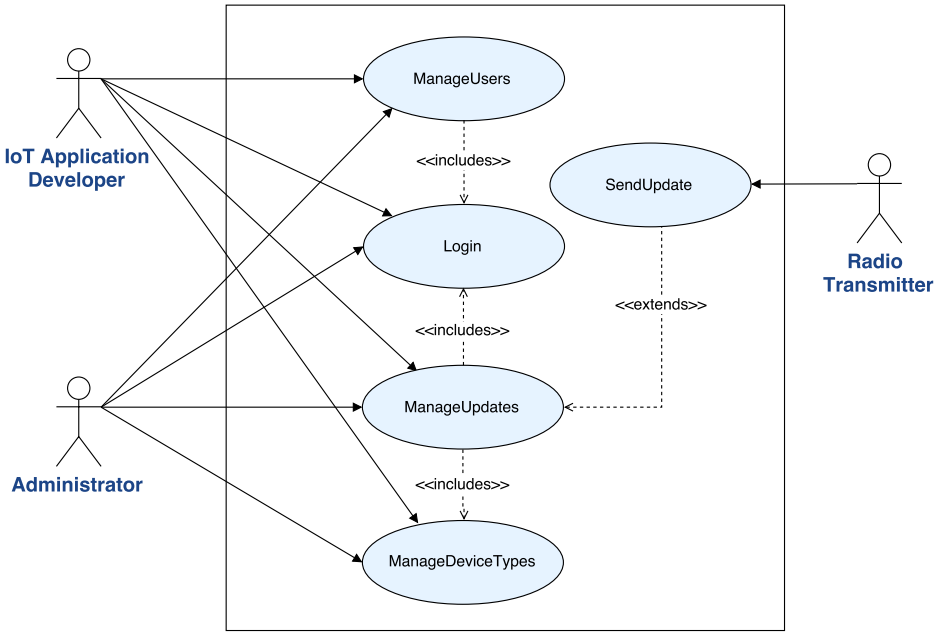


Figure 3.3: Use cases of the backend server.

developer and the administrator. Furthermore, an user authentication is required to perform any action.

- **Login.** Allows users to authenticate with the system and access the Web application functionalities.
- **ManageUpdates.** Provides functionalities to create, edit, view and delete updates. This use case is utilized by the IoT application developer and the administrator. Furthermore, a previous user authentication is necessary to execute any function. Additionally, it signs updates upon creation of the update package.
- **ManageDeviceTypes.** Offers functionalities such as create, edit, view and delete device types. These functionalities are utilized by the IoT application developer and the administrator, and these functionalities require the authentication of the user.
- **SendUpdate.** Runs in the backend server. Once the scheduled time of the update is reached, it sends the update towards the radio transmitter.

Figure 3.4 presents the following use cases of the IoT device:

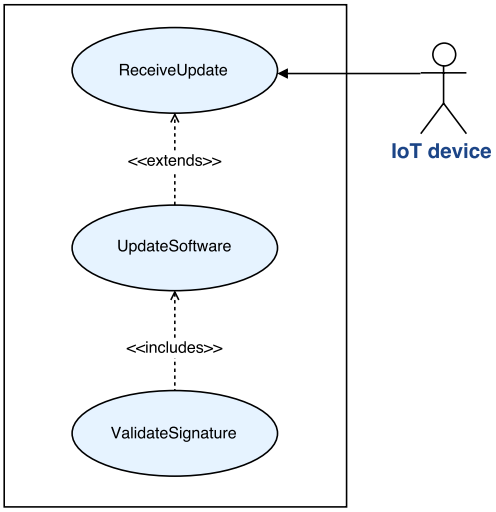


Figure 3.4: Use cases of an IoT device.

- **ReceiveUpdate.** Receives the data sent by the radio transmitter.
- **UpdateSoftware.** Processes the received image. Firstly, it includes the *ValidateSignature* use case to verify if the update package contains a valid signature. Afterwards, it runs the update installation process in the background by applying seamless updates process to install the OS update in the inactive system and reboot the devices after the installation. Additionally, it can update the applications installed at the IoT device without restarting the device.

3.2 Security Requirements

We now present the considered threat model so as to establish the security objectives of our solution. Threat modelling analyzes vulnerabilities in the system that can be exploited to perform malicious attacks, as well as mitigation measures to avoid these attacks [29]. Firstly, the model determines the assets of the system. Secondly, the model presents possible attackers and their motivation. Thirdly, the model specifies some attacks that could be performed in the system and the mitigation actions for each attack. Finally, the security goals of the system are established based on the model.

The assets of a system represent valuable components that are vulnerable

to attacks and could affect its proper operation. The identified assets of the IoT application provisioning service are:

- Backend Server
- IoT devices
- Image files (i.e., code of the update)
- Reputation of the service

Attacker	**Motivation**
Hacker	- Obtain the update code - Control IoT devices
Robber	- Obtain physical access to IoT devices
Malicious Employee	- Install malicious software into the IoT devices
Competitor Service	- Destroy the competition to gain clients for them

Table 3.1: Threat model: Attackers and their motivation.

Table 3.1 identifies the possible attackers of the system and their motivations. The threat model considers hackers as any intruder that interferes or extracts the data stored, transmitted or processed by the backend server or end devices. Additionally, the model includes the physical attacks that could be performed by a robber trying to steal the IoT device. Furthermore, the competitor service could pay employees to install malicious code into different nodes of the system to affect the reputation of the service and gain clients for them.

Attacks	**Mitigation**
Install malicious update	- Authenticate users - Only install valid signed updates
Obtain data during transmission	- Send encrypted data
Compromise backend server	- Harden the backend server security (i.e., with CloudHSM [39])
Compromise IoT devices	- Use TEE [5] to secure sensitive data

Table 3.2: Threat model: Attacks and mitigation measures.

Table 3.2 presents possible attacks to the system and their mitigation. The described attacks are explained in general terms. For instance, to avoid

Security goals	Description
Confidentiality	- Only authorized users should access sensitive data
Integrity	- Unauthorized users should not modify data in the system
Non-repudiation	- The author of the update cannot deny its authorship
Authentication	- Only authorized users are allowed to access the backend server - Verification of the author of the update in IoT devices
Authorization	- Only authenticated users are allowed to upload updates
Availability	- Update service should be running when authorized users require it

Table 3.3: Security goals of the IoT Applications Provisioning Service.

the installation of malicious software in the end devices, it is necessary to authenticate the users that are able to create the update packages and only install properly signed updates in the IoT device. Although the data encryption is performed prior to the transmission, it does not ensure complete confidentiality. In fact, unauthorized users can still receive such data. However, it prevents them from being able to understand the received information. In addition to these attacks, if the backend server or the IoT devices are compromised, it is important to harden the security of the server with CloudHSM and the end devices with platform security such as TEE. There are more specific attacks including brute force attack to guess the password of the user [47], man-in-the-middle (MITM) [48] to obtain or alter transmission of packets, as well as denial of service (DoS) attacks [49] that restrict the availability of the system affecting the reputation of the service. To mitigate these attacks is necessary to follow general security recommendations, e.g., allow only a certain number of login attempts per username, employ Hypertext Transfer Protocol over Transport Layer Security (HTTPS) [50] and analyze the requests rate in the server to detect any anomaly. Moreover, it is necessary to apply company rules to prevent that an employee releases a malicious update, e.g., the release process could require a prior approval of at least two employees. Additionally, the evaluation of the backend server implementation should be based on recommendations from About The Open Web Application Security Project (OWASP) about most critical risks on website applications [38].

The security goals of the system are described in Table 3.3. Based on this threat model and the scope of the thesis, we will focus on providing the system with authentication and integrity during the upgrade process. Furthermore, the system should offer confidentiality of the transmitted sensitive data.

3.3 HW/SW Architecture

The HW/SW architecture of the system is based on the previously introduced system design and its requirements. The system is independent of the wireless communication technology, hence we only consider two components: the backend server and the IoT device. Figure 3.5 shows the architecture of such components.

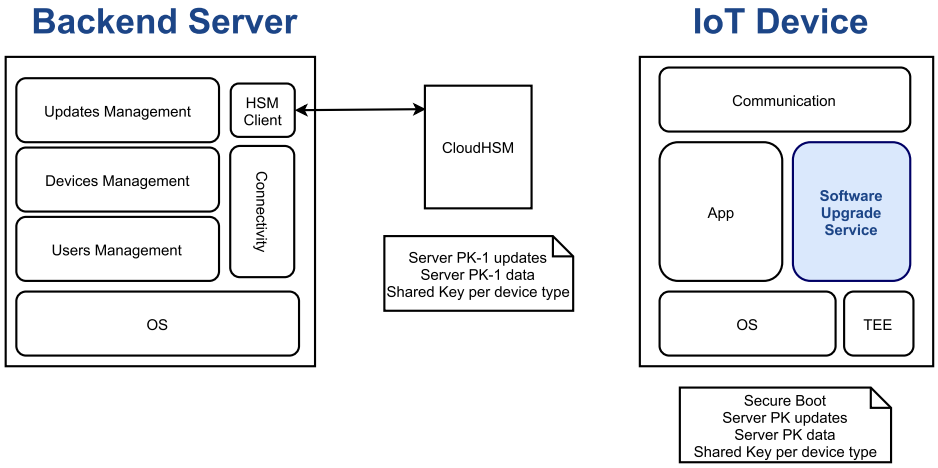


Figure 3.5: HW/SW Architecture of the IoT Application Provisioning Service.

The backend server includes an OS, where are implemented the functionalities to manage updates, devices and users. Moreover, the server requires a connectivity layer to connect to Internet to expose its services and send data towards the radio transmitter. Furthermore, the server possesses a hardware secure module client (HSMClient) that requests to the cloud hardware secure module (CloudHSM) to execute any sensitive operation of the system. The CloudHSM stores all the security keys. This architecture allows to migrate the backend server and utilize the same CloudHSM.

The IoT device runs an OS. Inside this OS, one or more applications could be installed in the system. Besides, a communication layer is required

to connect a radio broadcast receiver (e.g., RTL-SDR dongle) to obtain an update. Furthermore, a software upgrade service is installed to process the incoming updates and execute seamless updates. Additionally, a TEE is implemented in the device to store the security keys and run any trusted operations including the secure boot of the system.

Integrity requires that the backend server contains a private key to sign updates and the sent data. If several backend servers are involved, the CloudHSM stores as many keys as servers. However, one key is sufficient for one backend server (see Figure 3.5). Additionally, to provide confidentiality, a shared key is required between the backend server and the IoT device to encrypt sensitive information. The shared key is device type-specific and it is configured in the IoT devices by the manufacturer.

The presented architecture enables the creation of updates for a specific type of devices and its distribution through a wireless transmission technology to be received by the IoT devices. Such architecture allows seamless updates to update any partition in the hard disk of the IoT devices and provides higher availability time. However, the unidirectional communication provides no acknowledgement feature, hence the IoT devices cannot communicate with the backend server, reducing the reliability of the system. Additionally, the system design would compromise all the device types that share the same key if one of shared keys is exposed.

Chapter 4

Seamless Updates Prototype

This chapter details the implementation of the seamless update prototype. After an overview of the prototype, the chapter describes the Web application on the backend server. Furthermore, the chapter explains the development of the system upgrade service in the IoT device. Finally, the chapter evaluates the performance of the prototype, in terms of the time taken for the different phases of the update process at the embedded device.

4.1 Prototype Overview

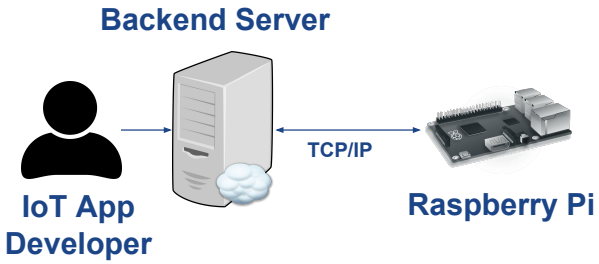


Figure 4.1: Prototype architecture.

The thesis work focuses on the implementation of the update service at the IoT device and the backend server, thus the long-range broadcast communication is not contemplated in the prototype. Figure 4.1 presents the architecture of the seamless update prototype. The IoT application developer uploads a device type-specific update to the backend server through a

Web application. The backend server publishes the software updates. The IoT device, i.e., Raspberry Pi in this prototype, requests the backend server for a new update for its device type. If a new version is found on the server, the IoT device downloads the update and performs seamless updates during the installation process. The prototype leverages TCP/IP as a communication protocol between the Raspberry Pi and the backend server. Such communication could be replaced in the future by a long-range broadcast technology with trivial modifications in code.

4.2 Web Application

The Web Application is implemented in a backend server. The server configuration includes:

- Linux-based OS, Ubuntu version 16.04
- Apache HTTP Server, version 2.4.18
- MySQL database, version 5.7.18
- PHP scripting language, version 7.0.18
- OpenJDK, version 1.8.0_131

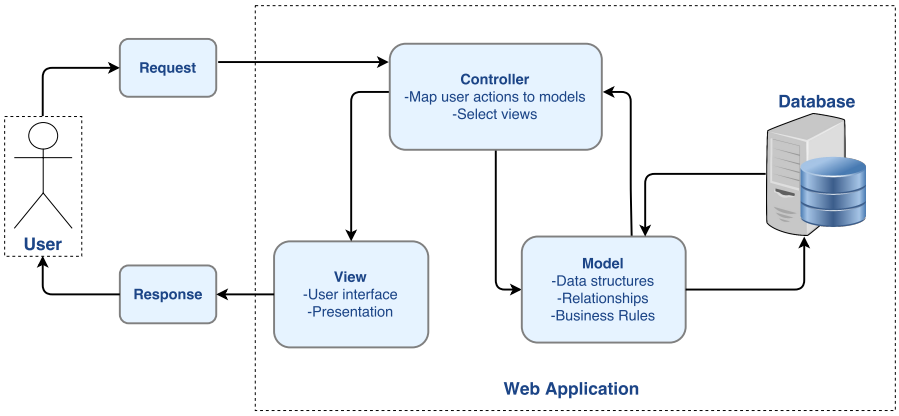


Figure 4.2: Model-View-Controller architecture [6, 7].

The application employs the CakePHP framework (version 3.4) to build the website. This framework leverages Model-View-Controller (MVC) architectural pattern [7] to facilitate software programming. Figure 4.2 shows how

CakePHP handles requests and responses of the user, as well as the communication between different components. To start, the user sends a request to the controller. Then, the controller calls to the models for information. Subsequently, models communicate with the database to obtain information and send it back to the controller. To finish the process, the controller invokes a view that presents the response to the user. Moreover, CakePHP offers several tools to apply security measures to the website, including the OWASP recommendations on most known vulnerabilities such as SQL injection and Cross-site scripting (XSS) prevention [38].

The website implements the use cases described in Figure 3.3 including:

- **Login.** Figure 4.3 presents the website to authenticate the user with username and password. The backend determines whether the user is an administrator or not, restricting the non-administrators from certain functionalities including the creation of users and the modification of information from other users.

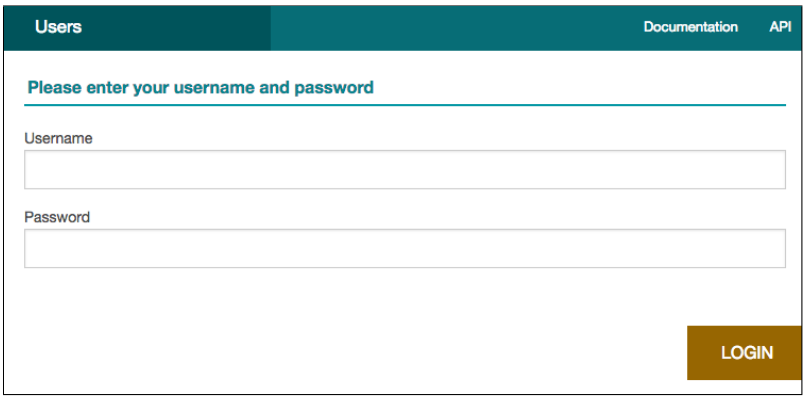


Figure 4.3: Login functionality.

- **Users Management.** Figure 4.4 shows the page that requests relevant information to create a new user: username, password, type (administrator or not) and access status (active or inactive). Only the administrator is able to create new users. Furthermore, normal users are allowed to only modify their information. One security measure implemented is to store the password hashed into the MySQL database instead of saving it as plain text. Furthermore, non-active users are unable to login into the system.

- **Device Types Management.** Figure 4.5 displays the website that requires only the name to store a new device type. All the users of

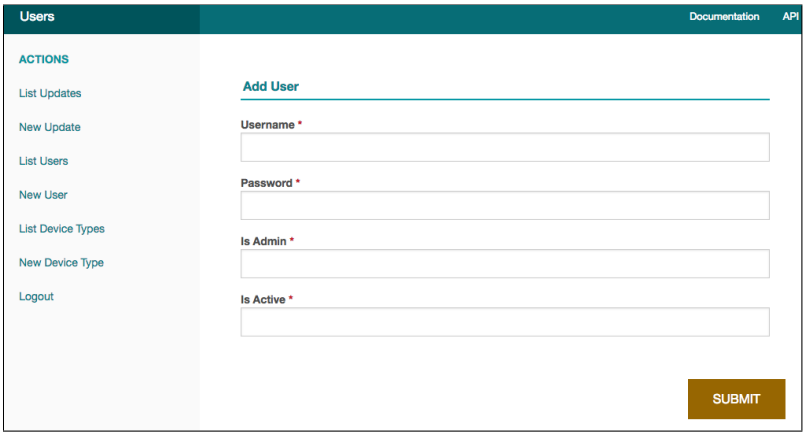


Figure 4.4: Add new user functionality.

the system are allowed to create device types, as well as to modify and remove these created device types. However, only administrators can modify and remove device types from other users.

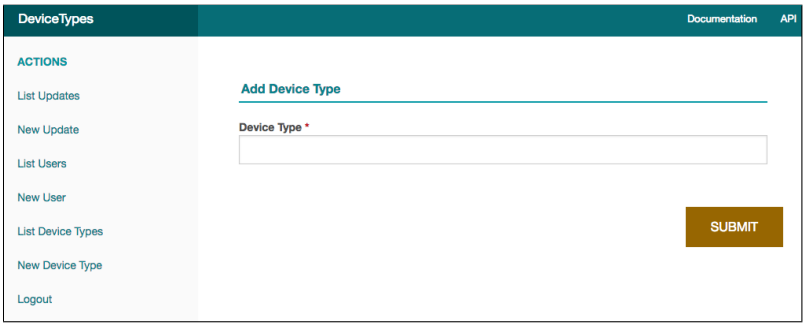


Figure 4.5: Add new device type functionality.

- **Updates Management.** Figure 4.6 shows the page to add a new update. The page requests the name, version, schedule time, device type and the file to create it. After the file is uploaded, the backend creates an update JAR file with the uploaded file and signs it with the private key of the server. The signature is performed with the jarsigner library [40]. This library is currently utilized by Android to sign their updates and supports three signing algorithms: DSA, RSA and ECDSA.
- **Send Update.** The prototype does not implement the functionality to send the update to the radio transmitter. However, the backend

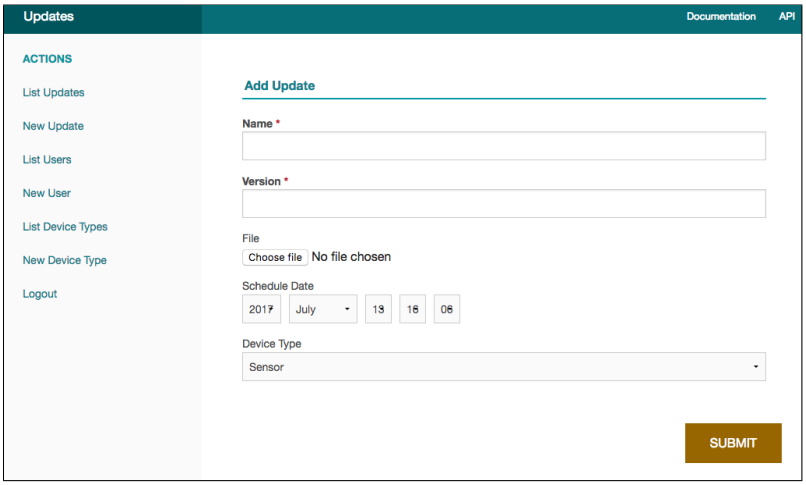


Figure 4.6: Add new update functionality.

allows updates to be available only after a scheduled time. To this end, it employs a cron job that runs every minute to trigger the Web application and search if there is any scheduled update to be released. If so, the newly schedule update is made available to the IoT devices for download.

4.3 IoT System Upgrade Service

For the prototype, the system upgrade service is implemented in a Raspberry Pi 3 (RPi3). The RPi3 enables the implementation of the HW and SW architecture presented in Figure 3.5. However, the utilized IoT device does not include a TEE. The configuration of the IoT device requires New Out of the Box Software (NOOBS) version 2.4 to install two OSes in the same disk. NOOBS acts as a bootloader, deciding which system is going to boot, i.e., either system A or B (see Subsection 2.2). These two systems are installed in the SD card along with NOOBS. The chosen Linux-based OS is Raspbian Lite (also known as Jessie) version 8 to save space in the Secure Digital (SD) memory card.

Additionally, the IoT device requires a desktop environment such as PIXEL (version 1.2) that provides a graphical user interface (GUI) for Raspbian Lite. PIXEL includes preinstalled applications for certain purposes including calendar, games and programming [51]. Finally, Oracle JDK version 1.8.0_65 is necessary to leverage jarsigner library to verify the signature of the received JAR file.

Table 4.1 shows all the partitions of the SD card[1]. Figure 4.7 presents a visual description of the partitions with system A and B specified to implement seamless updates. Each system A/B is an independent Raspbian Lite OS with their own boot and a root partitions. Typically, there is a maximum of four primary partitions on the SD card. For instance, this SD card is provided with only two, i.e., the RECOVERY partition in charge of the boot process and the extended partition. To allow more than four partitions, the extended partition contains the rest of the partitions as logical partitions. The SETTINGS partition contains all the configuration of NOOBS, e.g., which operating systems are installed and which OS is the default one [2]. Figure 4.8 displays the RPi3 running on system A and showing all the partitions of the SD card.

Primary partition	**Logical partition**	**Type**	**Label**	**Contents**
1		FAT	RECOVERY	NOOBS boot files & initramsfs[2], OS recovery images
2		extended		Any logical partitions
	5	ext4	SETTINGS	NOOBS settings
	6	FAT	boot	System A - Raspbian Lite boot files
	7	ext4	root	System A - Raspbian Lite filesystem
	8	FAT	boot0	System B - Raspbian Lite boot files
	9	ext4	root0	System B - Raspbian Lite filesystem

Table 4.1: SD card partitions [2].

The software of the system upgrade service is implemented as a shell script. The script has to search for the current version and device type on the IoT. Afterwards, it requests any available update in the Web server by providing the current version with a version greater than the current one. If it finds an update (i.e., a newer version of the device type), the update service downloads the file through Internet. Once the download is completed, the script verifies the certificate of the update file and the integrity of the JAR file through its signature. Then, it updates the relevant partition.

[1]The prototype utilized a 16GB card

[2]Initial RAM file system [52].

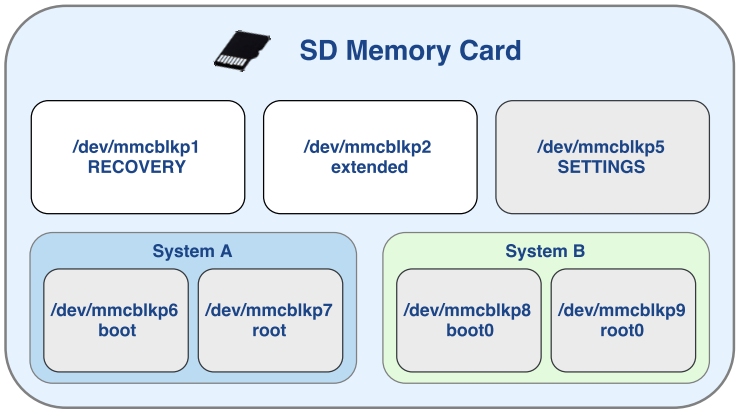


Figure 4.7: Structure of SD card with seamless update.

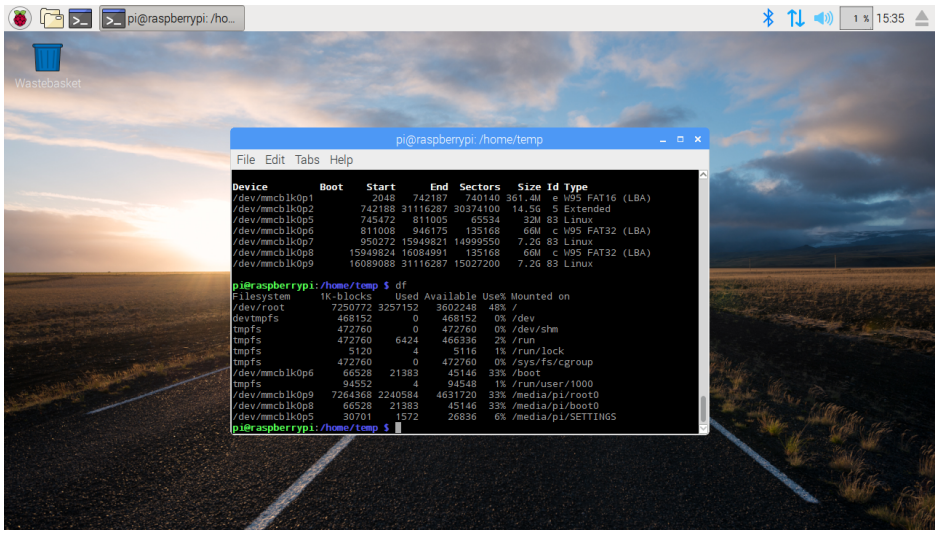


Figure 4.8: Raspberry Pi 3 partitions running in system A.

Once it finishes with the installation, it instructs the bootloader to boot the system with the newest version. Consequently, it requests the user to reboot the system, hence the device restarts and the bootloader tries to start the updated partition. In case the new partition is not bootable, the old version of the system is still accessible on the SD card and can be rolled back if needed. Additionally, if none of the systems are bootable, the NOOBS allows to reinstall the two OSes from an image saved in the RECOVERY partition.

Furthermore, an implementation with Linux frame buffer image viewer (fbi) package enables the prototype to display splash screen with the current

version of the device at the booting time before entering to the GUI [53]. Figure 4.9a shows the splash screen for a device with version 1.0. The prototype only considers incremental updates, i.e., from version 1.0 the device only upgrades to 2.0, it cannot skip any updates. This implementation could be modified in code to allow several upgrades in the same update package to install the latest version by executing only once the upgrading service.

To verify the device type and version of the update, the manifest in the JAR file provides the following information in the header:

```
Manifest-Version: 1.0
Device-Type: RaspberryPi3A_42
Update-Version: 2.0
Created-By: 1.8.0_131 (Oracle Corporation)

Name: 2.05.zip
SHA-256-Digest: vIlJ4fO8TWBahOI/13STSK1NPd23dUvxhvy2NZkuLW4=
```

Listing 4.1: Manifest file for system upgrade service

Figure 4.9 demonstrates the phases to execute the seamless update prototype. Firstly, the RPi3 boots and displays as a splash screen Figure 4.9a to show the current version 1.0. Secondly, Figure 4.9b displays highlighted the current boot partition *mmcblk0p6*, thus the device is running system A. Moreover, the output of the system upgrade shell script is presented in Figure 4.9c. As previously mentioned, the script downloads, verifies, installs the new boot version and requests the user to reboot the IoT device. Consequently, the RPi3 reboots after the software update and show Figure 4.9d as splash screen with the updated version 2.0. Finally, the IoT device swaps to run system B with the latest installed version. As expected, Figure 4.9e highlights the new boot partition *mmcblk0p8*, hence the upgrade is successful and the new version is properly running on system B.

This system upgrade service runs in the background, thus it allows the user to continue working with the IoT device. The seamless updates implementation enables to update any partition of the system and provides more availability time of the devices, removing the unavailability of the IoT device during the installation time.

4.4 Evaluation

To evaluate the prototype implementation, the system upgrade service is executed five times in the RPi3 with the *time* command to obtain the real time to run the script, then determine its average performance. Time is the only considered metric in the evaluation. The device implements seamless

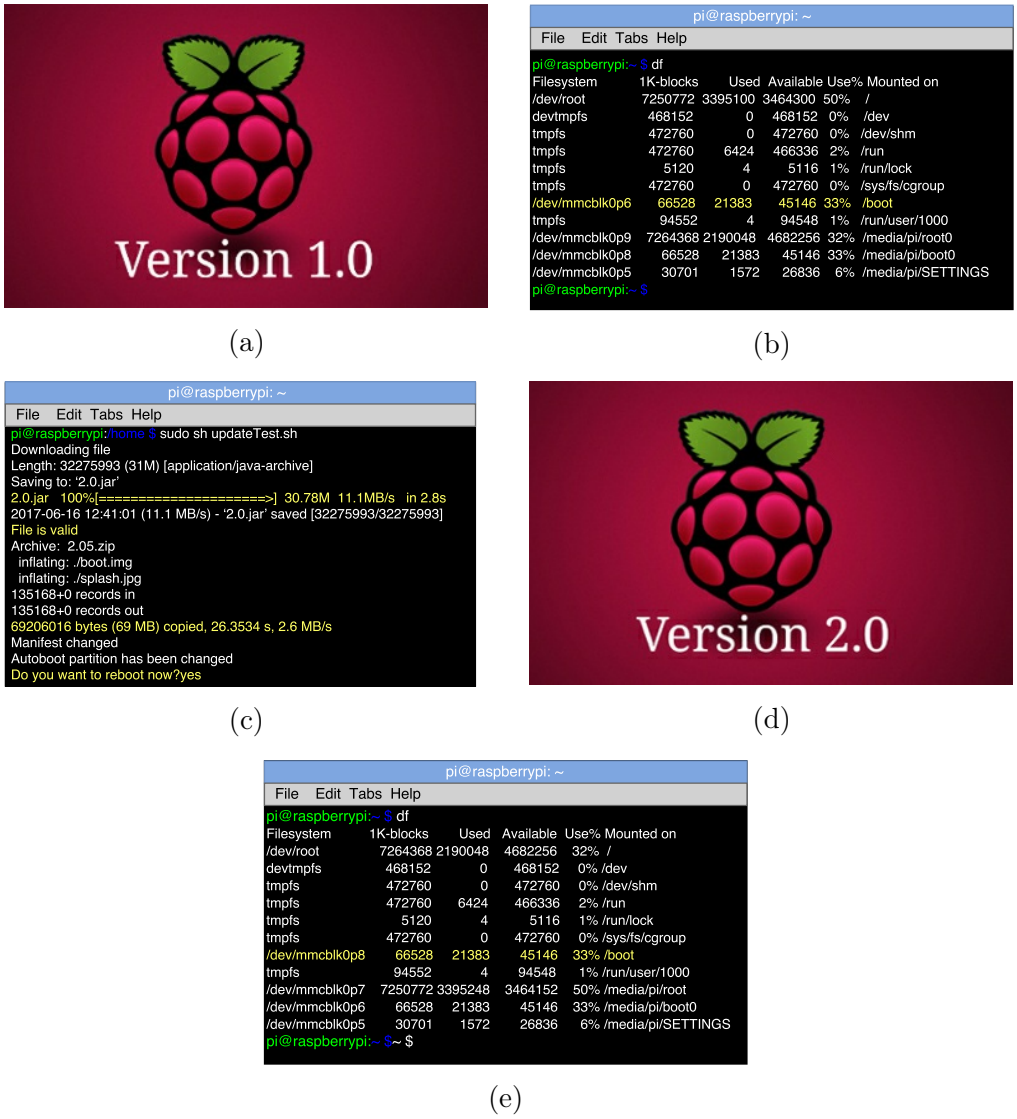


Figure 4.9: Example of the system upgrade service. a) Splash screen of an IoT device with version 1.0. b) Console output of the FS of a device running in system A. c) System upgrade service console output. d) Splash screen of an IoT device with version 2.0. e) Console output of the FS of a device running in system B.

updates, hence the installation is performed in the background. The evaluation includes the time to download, to write to the inactive A/B system and the rest of actions to accomplish the update: the validation of the signature, the modification of the boot files, as well as the update of the splash screen. Table 4.2 shows the time to install the update in the IoT device[3] and Table 4.3 presents the performance of the system upgrade service as a whole, specifying the total time of installation with rebooting time. The results of the evaluation demonstrate that the IoT device is updated in less than a minute and a half whereas the total unavailability time is approximately 36 seconds thanks to the implementation of seamless updates.

Functionality	**Min. time (seconds)**	**Max. time (seconds)**	**Average time (seconds)**
Download	2.8	2.8	2.8
Write in partition	21.69	22.21	21.99
Rest of the script	18.26	21.72	20.17
Installation Time	42.75	46.73	44.97

Table 4.2: Performance in installation process.

Functionality	**Min. time (seconds)**	**Max. time (seconds)**	**Average time (seconds)**
System Upgrade Service	42.75	46.73	44.97
Reboot	32.67	37.43	35.25
Total Time	75.42	84.16	80.22

Table 4.3: Performance in system upgrade service.

Another important aspect is the performance of verifying digital signatures at the IoT device. Our experiments involve signing several JAR files with different parameters including the signing algorithm, the digest hash algorithm and the key size. Moreover, each JAR file is verified four times to obtain the verification time. The metrics of the evaluation involve the time to verify the file and the size of the files after being signed with the different parameters. More specifically, to define the signing parameters it is necessary to follow the security recommendations and standards on digital signatures [54], key management [55] and secure hashes [56] of the US National Institute of Standards and Technology (NIST), as well as the specifications and restrictions of Oracle JDK [57]. For instance, NIST recommends

[3]The update size is 32.3 MB

the usage of ECDSA in IoT devices because of its performance and efficiency, i.e., it provides higher security[4] than RSA/DSA with smaller keys. For this reason, ECDSA is evaluated utilizing two different key sizes 224 and 571 to analyze how its performance is affected by the size of the key. Additionally, the RSA is evaluated with a key size of 2048 to provide a similar security as the 224 key of ECDSA [58]. Typically, the maximum key size of DSA is 1024 based on the restrictions established by Oracle JDK [57]. Furthermore, Oracle follows NIST recommendations, hence the JAR file is not signed if the specified parameters do not follow the security standards, or if they do not comply with key size restrictions of Oracle.

Signing Algorithm	**Key Size**	**File size (bytes)**	**Verification Time on RPi3 (seconds)**
DSA	1024	63324572	4.913
RSA	2048	63324817	4.902
ECDSA	224	63324232	4.91
ECDSA	571	63324505	5.205

Table 4.4: Experiments performed for update files using SHA-1 digest algorithm.

The first group of experiments are performed utilizing SHA-1 as digest hash algorithm. Table 4.4 shows the performance results for each signing algorithm and key size. Figure 4.10 also presents the measured times. The difference between the DSA(1024), RSA(2048) and ECDSA(224) is less than 10 milliseconds. Only ECDSA(517) increased the time difference due to a key size which provides higher security than the other signing options. As expected, for one algorithm, the time increases with the key size. The verification time is the lowest than the other two digest algorithms, SHA-224 and SHA-256. However, the collisions found in SHA-1 represent vulnerabilities on the algorithm. For this reason, NIST recommends to utilize SHA-2 for new applications.

Additionally, a second group of experiments leverages SHA-224 digest algorithm with different signing algorithms. Table 4.5 presents time results of the experiments, while Figure 4.10 shows the time difference. The evaluated time difference is a little more than 10 milliseconds comparing DSA(1024) to RSA(2048) and ECDSA(224). As previously mentioned, ECDSA(571) time is higher than the other three cases. This SHA-224 digest algorithm is recommended by NIST for applications from 2016 until 2030 [58].

[4] *Higher security* means that is less vulnerable or increases the complexity to be hacked.

Signing Algorithm	Key Size	File size (bytes)	Verification Time on RPi3 (seconds)
DSA	1024	63324609	6.1935
RSA	2048	63324861	6.16575
ECDSA	224	63324283	6.169625
ECDSA	571	63324556	6.42435

Table 4.5: Experiments performed signing update files using SHA-224 digest algorithm.

Signing Algorithm	Key Size	File size (bytes)	Verification Time on RPi3 (seconds)
DSA	1024	63324626	6.20275
RSA	2048	63324858	6.1705
ECDSA	224	63324292	6.17775
ECDSA	571	63324566	6.43475

Table 4.6: Experiments performed for update files using SHA-256 digest algorithm.

The last set of experiments is performed with SHA-256 digest hash algorithm. Table 4.6 presents the measured times. The obtained values are the highest of the three sets of experiments, as SHA-256 requires more time to be executed than SHA-1 and SHA-224. Figure 4.10 demonstrates the behavior of the signing algorithms, which is very similar comparing the three groups of experiments. Furthermore, analyzing the size of the three signing algorithms, ECDSA files are smaller in bytes than with the other two.

The evaluation demonstrates that the performance is affected when security is increased, i.e., SHA-1 is faster but provides less security than SHA-2 algorithms. Similarly, the verification time increases with the key size of ECDSA. Moreover, ECDSA offers at least the same level of security as RSA or DSA with smaller keys. As a result, the size of their signed files is smaller than the other two signing algorithms. For future IoT applications, the necessity to increase the security with a lower amount of resources demands ECDSA as a more suitable solution for embedded devices.

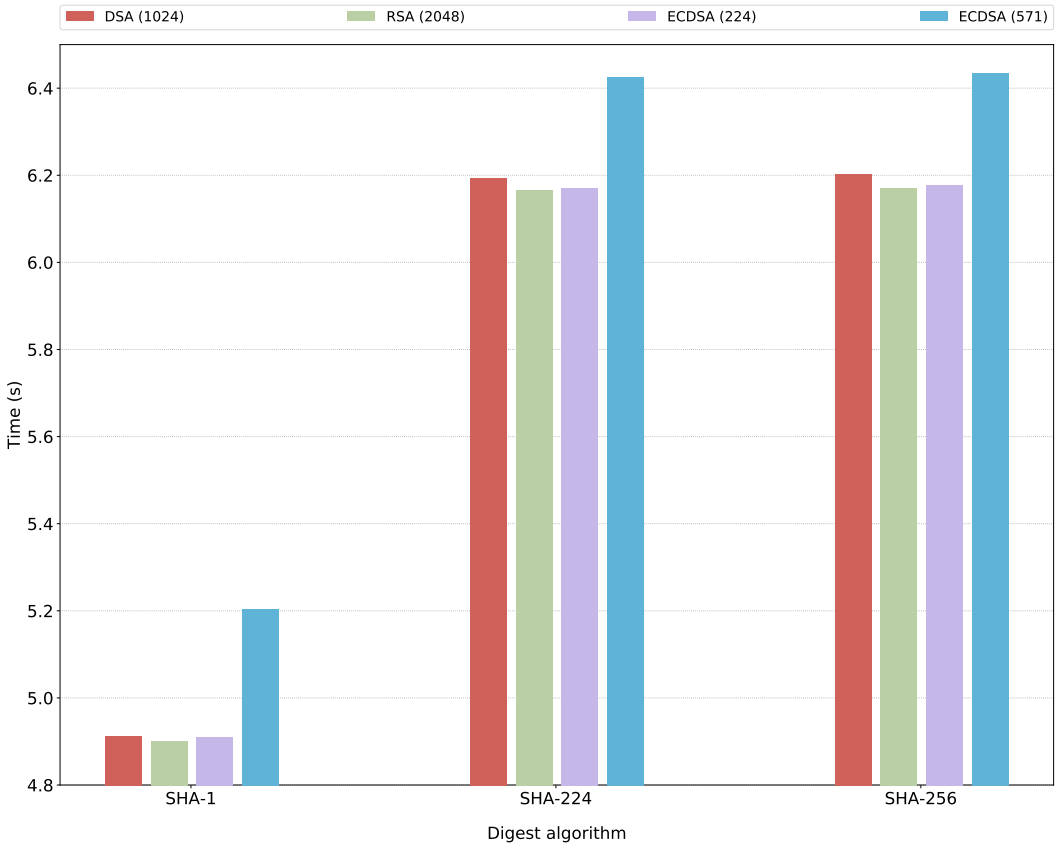


Figure 4.10: Comparing the performance of different signing parameters.

Chapter 5

Conclusion

This thesis presented an architectural solution that provides software updates for IoT devices leveraging long-range broadcast communications. The reference scenario and threat model provided requirements of the system in terms of components, protocols and security goals. Additionally, a seamless updates prototype was implemented for an IoT device based on the solution in ChromeOS/Android to extend the availability time of the devices during the updating process and upgrade any partition of their system. Furthermore, the broadcast provisioning service groups IoT devices by different device types to distribute and manage their software updates in the Backend Server through a Web application.

The design of the solution is compliant with security measures on IoT devices about key sizes, signing algorithms and cryptographic protocols. In addition to these recommendations, the designed structure of the IoT device includes a TEE to provide platform security, while the backend server design implements a CloudHSM to execute any sensitive functionality and store private data. However, it is important to take into account the most known vulnerabilities of OWASP on Web Applications such as SQL injection and XSS in the backend implementation.

The proposed IoT Application Provisioning Service does not rely on a specific communication as long as it is long-range broadcast digital data. Although the implemented prototype requires Internet connection on the IoT device, it could be easily modified to leverage any long-range broadcast communication technology. Consequently, the system represents a cheaper option to provide software updates to a wide area in terms of infrastructure than other cellular or multi-hop short-range wireless communication technologies.

The project could require a scalable Cloud solution on the backend to handle a large amount of IoT application developers and embedded devices. Other improvements include leveraging incremental cryptography, i.e., the

idea is that once the hash value of a file is computed, it is not required to recompute it entirely if the file is modified, hence executing the hash algorithm only in the modified section of the file [59]. Incremental cryptography enables faster cryptographic operations increasing the efficiency of the systems, hence it could be applied on incremental updates to avoid recomputing the entire files in the update package. Besides, to increase the reliability of the wireless transmission, fountain codes or Reed-Solomon codes could be implemented to mitigate any packet loss while transmitting data to the embedded devices [60]. In addition, a multicast encryption could be implemented to improve the confidentiality of the system. Multicast encryption would allow only a group of end devices to access the sent update image. However, the current multicast encryption solutions require rekeying functionality to update the shared key of the members of the group each time a member enters or exits the group. According to the proposed provisioning service, only unidirectional communication is present on the system, thus there could not be a confirmation if all the devices received the new key and modified their old one. For this reason, this type of key management functionality on a large scale of devices would not be suitable on an urban scenario and it is not feasible for the current assumptions of the presented IoT Application Provisioning Service [61, 62].

As a future work, the solution should be modified to utilize a long-range broadcast communication technology in a real environment. The seamless updates prototype should be extended to support other types of embedded devices including Edison Board, Arduino and Banana Pi, as well as to implement several functionalities including the rollback process, the number of remaining boot attempts and the encryption of the updates. Moreover, the system could update the device directly to the latest update (i.e., instead of incremental updates) and the IoT device should compress the size of the file system to save space in the hard disk. All these modifications will increase the efficiency of the presented solution.

Bibliography

[1] S. Verma and G. S. Prajapati, "Robustness and security enhancement of SHA with modified message digest and larger bit difference," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pp. 1–5, March 2016.

[2] "NOOBS partitioning explained." `http://github.com/raspberrypi/noobs/wiki/NOOBS-partitioning-explained`. Accessed : Jul 14, 2015.

[3] Chronium OS, "File system/autoupdate." `http://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate`. Accessed : Feb 1, 2017.

[4] Chronium OS, "File system/autoupdate supplements." `http://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate-supplements`. Accessed : Feb 1, 2017.

[5] ARM Ltd., "TrustZone." `http://developer.arm.com/technologies/trustzone`. Accessed : Mar 02, 2017.

[6] J. Plekhanova, "Evaluating web development frameworks: Django, Ruby on Rails and CakePHP," *Institute for Business and Information Technology*, 2009.

[7] *CakePHP Application Development, author=Bari, Ahsanul and Syam, Anupom, year=2008, publisher=Packt Publishing Ltd.*

[8] J. Carter, E. Grommon, and P. Harris, ""From conceptual to operational: Over-the-air-programming of land mobile radios"," *Physical Communication*, vol. 19, pp. 18 – 29, 2016.

[9] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, "Long-range communications in unlicensed bands: the rising stars in the IoT and smart city scenarios," *IEEE Wireless Communications*, vol. 23, pp. 60–67, October 2016.

[10] L. C. P. Costa, R. A. Herrero, M. G. D. Biase, R. P. Nunes, and M. K. Zuffo, "Over the air download for digital television receivers upgrade," *IEEE Transactions on Consumer Electronics*, vol. 56, pp. 261–268, February 2010.

[11] Android, "OTA Updates." `http://source.android.com/devices/tech/ota/`. Accessed : March 28, 2017.

[12] Android, "Block-Based OTAs." `http://source.android.com/devices/tech/ota/block`. Accessed : Feb 14, 2017.

[13] S. Unterschütz and V. Turau, "Fail-safe over-the-air programming and error recovery in wireless networks," in *Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems*, pp. 27–32, July 2012.

[14] Android, "A/B System Updates." `http://source.android.com/devices/tech/ota/ab_updates`. Accessed : Feb 14, 2017.

[15] M. C. Artemiy I. Pavlov, "Squash FS HowTo." `http://tldp.org/HOWTO/SquashFS-HOWTO/index.html`. Accessed : Feb 3, 2017.

[16] I. F. Akyildiz, W.-Y. Lee, M. C. Vuran, and S. Mohanty, "A survey on spectrum management in cognitive radio networks," *IEEE Communications magazine*, vol. 46, no. 4, 2008.

[17] S. Kawade and M. Nekovee, "Is wireless broadband provision to rural communities in TV whitespaces viable? A UK case study and analysis," in *2012 IEEE International Symposium on Dynamic Spectrum Access Networks*, pp. 461–466, Oct 2012.

[18] FCC, "Notice of proposed rule making and order." `http://apps.fcc.gov/edocs_public/attachmatch/FCC-03-322A1.pdf`, December 2003.

[19] W. Lehr, "Toward more efficient spectrum management, new models for protected shared access. Technical report, MIT Communications Future Program." `http://cfp.mit.edu/publications/CFP_Papers/CFP%20Spectrum20Sharing%20Paper%202014.pdf`.

[20] C. W. Pyo, X. Zhang, C. Song, M. T. Zhou, and H. Harada, "A new standard activity in IEEE 802.22 wireless regional area networks: Enhancement for broadband services and monitoring applications in TV whitespace," in *The 15th International Symposium on Wireless Personal Multimedia Communications*, pp. 108–112, Sept 2012.

[21] T. X. Brown and D. C. Sicker, "Can Cognitive Radio Support Broadband Wireless Access?," in *2007 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*, pp. 123–132, April 2007.

[22] S. Bayhan, G. Premsankar, M. Di Francesco, and J. Kangasharju, "Mobile Content Offloading in Database-Assisted White Space Networks.," in *CrownCom*, pp. 129–141, 2016.

[23] S. Bayhan, L. Zheng, J. Chen, M. D. Francesco, J. Kangasharju, and M. Chiang, "Improving cellular capacity with white space offloading," in *2017 15th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, pp. 1–8, May 2017.

[24] Oona Räcisänen, "Broadcast messages on the darc side." `http://www.windytan.com/2013/11/broadcast-messages-on-darc-side.html`. Accessed : Jan 14, 2017.

[25] "About RTL-SDR." `http://www.rtl-sdr.com/about-rtl-sdr/`. Accessed : Jun 13, 2017.

[26] O. Banaei and S. Khorsandi, "A new quantitative model for web service security," in *2012 IEEE 14th International Conference on Communication Technology*, pp. 749–755, Nov 2012.

[27] N. Aschenbruck, J. Bauer, J. Bieling, A. Bothe, and M. Schwamborn, "Selective and Secure Over-The-Air Programming for Wireless Sensor Networks," in *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–6, July 2012.

[28] J. Gilger and H. Tschofenig, "Report from the smart object security workshop," 2014.

[29] P. H. Meland, E. A. Gjære, and S. Paul, "The Use and Usefulness of Threats in Goal-Oriented Modelling," in *2013 International Conference on Availability, Reliability and Security*, pp. 428–436, Sept 2013.

[30] D. Thaler, H. Tschofenig, and M. Barnes, "Architectural Considerations in Smart Object Networking," 2015.

[31] F. Koeune and F.-X. Standaert, "A tutorial on physical security and side-channel attacks," in *Foundations of security analysis and design III*, pp. 78–108, Springer, 2005.

[32] S. M. Bellovin and R. Housley, "Guidelines for cryptographic key management," in *Symposium on Research in Security and Privacy*, 2005.

[33] D. Eastlake 3rd, J. Schiller, and S. Crocker, "Randomness requirements for security," tech. rep., 2005.

[34] "Recommendations for secure use of transport layer security (TLS) and datagram transport layer security (DTLS), author=Sheffer, Yaron and Holz, Ralph and Saint-Andre, Peter, year=2015," tech. rep.

[35] H. Tschofenig, J. Arkko, D. Thaler, and D. McPherson, "Architectural considerations in smart object networking," tech. rep., 2015.

[36] Y. Sheffer, R. Holz, and P. Saint-Andre, "Summarizing known attacks on transport layer security (tls) and datagram tls (dtls)," tech. rep., 2015.

[37] A. Cooper, H. Tschofenig, B. Aboba, J. Peterson, J. Morris, M. Hansen, and R. Smith, "Privacy considerations for Internet protocols," tech. rep., 2013.

[38] OWASP, "Top 10." `https://www.owasp.org/index.php/Top_10_2013-Top_10`. Accessed : Jul 03 , 2017.

[39] Amazon Web Services, Inc., "AWS CloudHSM." `http://aws.amazon.com/cloudhsm/`. Accessed : Mar 02, 2017.

[40] Oracle, "Jarsigner." `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html`. Accessed : May 15, 2017.

[41] A. Sghaier, M. Zeghid, and M. Machhout, "Fast hardware implementation of ECDSA signature scheme," in *2016 International Symposium on Signal, Image, Video and Communications (ISIVC)*, pp. 343–348, Nov 2016.

[42] M. Silaghi, K. Alhamed, and R. Stansifer, "Java Tool Extensions for Supporting Multiple Recommenders and Distributed Bundles," in *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 722–725, Dec 2015.

[43] CENELEC European Standard, "Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87,5 to 108,0 MHz," 1998.

[44] M. Charbonneau-Lefort and M. J. Yadlowsky, "Optical Cables for Consumer Applications," *Journal of Lightwave Technology*, vol. 33, pp. 872–877, Feb 2015.

[45] A. Bit, M. Orehek, and W. Zia, "Comparative analysis of Bluetooth 3.0 with UWB and Certified Wireless-USB protocols," in *2010 IEEE International Conference on Ultra-Wideband*, vol. 2, pp. 1–4, Sept 2010.

[46] Specification, O M G Available and Bars, Change, "OMG Unified Modeling Language (OMG UML)," *Language*, no. November, pp. 1 – 212, 2007.

[47] J. Jose, T. T. Tomy, V. Karunakaran, A. K. V, A. Varkey, and N. C. A., "Securing passwords from dictionary attack with character-tree," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 2301–2307, March 2016.

[48] M. Conti, N. Dragoni, and V. Lesyk, "A Survey of Man In The Middle Attacks," *IEEE Communications Surveys Tutorials*, vol. 18, pp. 2027–2051, thirdquarter 2016.

[49] M. Ficco and F. Palmieri, "Introducing Fraudulent Energy Consumption in Cloud Infrastructures: A New Generation of Denial-of-Service Attacks," *IEEE Systems Journal*, vol. 11, pp. 460–470, June 2017.

[50] S.-W. Han, H. Kwon, C. Hahn, D. Koo, and J. Hur, "A survey on MITM and its countermeasures in the TLS handshake protocol," in *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 724–729, July 2016.

[51] "Raspbian Lite with PIXEL/LXDE/XFCE/MATE/i3 GUI." `http://www.raspberrypi.org/forums/viewtopic.php?f=66&t=133691`. Accessed : Jun 14, 2017.

[52] "Chapter 5. File Systems and Disk Management. About initramfs." `http://www.linuxfromscratch.org/blfs/view/svn/postlfs/initramfs.html`. Accessed : Jun 14, 2015.

[53] "Custom boot up screen." `http://developer.android.com/guide/topics/graphics/opengl.html`. Accessed : May 30, 2017.

[54] C. F. Kerry and P. D. Gallagher, "Digital signature standard (DSS)," *FIPS PUB*, pp. 186–4, 2013.

[55] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management part 1: General (revision 3)," *NIST special publication*, vol. 800, no. 57, pp. 1–147, 2012.

[56] F. PUB, "Secure Hash Standard (SHS)," *FIPS PUB 180*, vol. 4, 2012.

[57] "Keysize Restrictions." `http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html#importlimits`. Accessed : Jun 15, 2017.

[58] "NIST recommendations." `https://www.keylength.com/en/4/`. Accessed : Jun 13, 2017.

[59] H. Mihajloska, D. Gligoroski, and S. Samardjiska, "Reviving the Idea of Incremental Cryptography for the Zettabyte era Use case: Incremental Hash Functions Based on SHA-3," in *International Workshop on Open Problems in Network Security*, pp. 97–111, Springer, 2015.

[60] L. Pelusi, A. Passarella, and M. Conti, "Encoding for Efficient Data Distribution in Multihop Ad Hoc Networks," *Algorithms and Protocols for Wireless and Mobile Ad hoc Networks*, p. 87, 2009.

[61] D. Wallner, E. Harder, and R. Agee, "Key management for multicast: Issues and architectures," tech. rep., 1999.

[62] Y. Wu, J. Liu, J. Hou, and S. Yao, "A stateful multicast key distribution protocol based on identity-based encryption," in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, pp. 19–24, May 2017.